Project Threads Report

Group 43

Name	Autograder Login	Email
Sarvagya Somvanshi	student176	sarvagya@berkeley.edu
Nathan Canera	student282	nathancanera@berkeley.edu
John Chang	student271	jchang04@berkeley.edu
Shengxiang Lin	student35	linshengxiang@berkeley.edu

Changes

Efficient Alarm Clock

The Efficient Alarm Clock pretty much followed the design document, as we used the `sleeping_list` to manage all the sleeping threads ordered by wake times.

We calculated `wake_time` = `timer_ticks()` + `ticks` and then inserted that into the `sleeping_list` using the build in `list_insert_ordered` using the comparator we generated.

We use the `timer_sleep` function to block the thread until its wake time, while the `timer_interrupt` handler iterates through the `sleeping_list`, unblocking threads whose wake time has arrived. No significant changes were made to the original design.

Strict Priority Scheduler

Our actual implemntation of Strict Priority Scheduler was very similar to the design document. Our TA also gave us the suggest of disabling and enabling interrupts to ensure critical sections are not interrupted. Priority inheritance was implemented to prevent priority inversion, and we used `update_priority` and `recurse_donate` to allow threads to donate their `effective_priority` to others holding locks they need. The `priority_queue` was maintained as a sorted list, ensuring O(1) access to the highest-priority thread while adding and removing threads efficiently using

`list_insert_ordered` and a thread priority comparator, ensuring the list is painted as sorted.

Preemption was enforced by calling `intr_yield_on_return` in the timer interrupt as our TA suggested and during relevant operations like `thread_create`, `sema_up`, and `lock_release`, ensuring that the highest-priority thread always runs. This approach guarantees strict priority ordering and responsiveness, meeting the project's scheduling requirements.

User Threads

We can start our changes to the User Threads portion of project 2 with our changes to the process struct and the thread struct.

In our process struct, we removed the `next_tid` attribute and the `c_status` struct We removed `next_tid` because `thread_create` keeps track of what tid's are available so we don't have to. We removed `c_status` because all we needed was a semaphore in the `pthread_execute` function.
We also slightly adjusted the freedom struct to instead by a pointer to a struct which included an array of free pages and a lock for synchronization.

Our thread struct was slightly adjusted to simplify joining. Instead of storing our join struct `thread_joins` in each thread, we put a pintos list of them into the process struct so iteration is easy. The `thread_joins` struct contains `join_elems` which have a `tid` for the thread who owns it, a boolean `joined` that tells threads whether or not the thread who owns that struct has been joined on already, and a semaphore `join` for joining functionality.

We also didn't add an explicit pcb lock and instead used our lock in the pages struct for the synchronization necessary when manipulating the freed pages.

When exiting, we kept a lot of our implementation, but replaced the exit_status condition variable with a boolean `marked_for_slaughter`. When we reached the interrupt handler we would check if it was true and if it was pthread_exit was called.

Reflection

This new project was a step up from the last project as it challenged us to build upon the skills and collaborative approaches we honed during our previous work, while the new main components—Efficient Alarm Clock, Strict Priority Scheduler, and User Threads—demanded a careful balance of technical understanding and teamwork, and we were thrilled with how effectively we came together to meet these demands.

Implementing the **Efficient Alarm Clock(done by Shengxiang)** and **Strict Priority Scheduler(by Sarvagya)** was mostly straightforward due to our adherence to the design document. However, **User Threads(shared between Nathan and John)** components were more complex and required careful planning. The user threads portion required structural changes to simplify joining and memory management which genuinely challenged all of us during the debugging part. By collaborating effectively, we overcame challenges and refined our solutions through shared debugging and discussion, such as when debugging `multi-oom` where everyone's code and freeing ability was tested

---

Testing

a**larm-delay**: We test the use of the timer by multiple threads simultaneously by checking the accuracy of the timer delay. Specifically, we verify the accuracy of the timer delay by creating multiple threads, each hibernating for a specific period of time, and then checking the difference between the actual wakeup time and the expected wakeup time.

```
Expected:
(alarm-delay) begin
(alarm-delay) Thread with delay 100 (expected delay was 100, actual delay was 100).
(alarm-delay) Thread with delay 200 (expected delay was 200, actual delay was 200).
(alarm-delay) Thread with delay 300 (expected delay was 300, actual delay was 300).
(alarm-delay) Thread with delay 400 (expected delay was 400, actual delay was 400).
```

```
(alarm-delay) Thread with delay 500 (expected delay was 500, a
ctual delay was 500).
(alarm-delay) Thread with delay 600 (expected delay was 600, a
ctual delay was 600).
(alarm-delay) Thread with delay 700 (expected delay was 700, a
ctual delay was 700).
(alarm-delay) Thread with delay 800 (expected delay was 800, a
ctual delay was 800).
(alarm-delay) Thread with delay 900 (expected delay was 900, a
ctual delay was 900).
(alarm-delay) Thread with delay 1000 (expected delay was 1000,
actual delay was 1000).
(alarm-delay) end
```

Output and Results

When the test was run on our implemented Pintos kernel, the output matched the expected results, confirming that the `timer_sleep` function operates accurately for multiple threads with varying delays. The `.output` and `.result` files show no errors or unexpected deviations in delay times.

Non-Trivial Potential Kernel Bugs

1. **Incorrect Timer Tick Update:** If the timer interrupt handler (`timer_interrupt`) failed to update the timer ticks accurately (e.g., incrementing ticks by 2 instead of 1), the delay calculations would be skewed. This would cause threads to wake up earlier than expected, leading to an output like:

```
(alarm-delay) Thread with delay 100 (expected delay was 100, a
ctual delay was 50).
(alarm-delay) Thread with delay 200 (expected delay was 200, a
ctual delay was 100).
```

Indicating that the timer is unreliable and fails to respect requested delays.

2. **Improper Management of `sleeping_list`:** If the `sleeping_list` was not correctly sorted by wake times (e.g., threads added in random order without sorting), threads might wake up out of order. This means that thread with a

shorter delay might wake up after threads with longer delays, resulting in output like:

```
(alarm-delay) Thread with delay 200 (expected delay was 200, actual delay was 300).
(alarm-delay) Thread with delay 100 (expected delay was 100, actual delay was 100).
```

Indicating a flaw with the scheduling mechanism.

Writing test cases, we learned the importance of covering edge cases and it made us think more about how our projet works all together, deepening our understanding of PintOS and ultimitely benefiting our ability to write code in the future.